A transactional approach for cross-organizational cooperation

Manuel Munier, Khalid Benali, Claude Godart

A Transactional Approach for Cross-Organizational Cooperation

M. Munier and K. Benali and C. Godart
LORIA, UMR n°7503 - Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex - FRANCE
E-mail: godart@loria.fr

Abstract

Due to the popularization of Internet, virtual enterprises are expected to become commonplace on the WEB. The concept of virtual enterprise depicts the idea that many applications are the result of cross-organisational cooperation between several actors, playing different roles, who build a relational system which is structured by a common objective. Virtual enterprises can be short- or long-lived. The objective of our work is to develop a framework to install and to support cooperation betwen the partners of a virtual enterprise, with a particular focuss on concurrent engineering applications, and especially on co-design activities. We have chosen a transactional approach which releases programmers from the burden of interaction programming, and this approach has been implemented in a distributed manner using a cooperation model requesting a partner to negotiate cooperation patterns with its direct neighbours.

Keywords: **cooperation, distributed systems, transaction models**

I. Introduction

Due to the popularization of Internet, *virtual enterprises* are expected to become commonplace on the WEB. The concept of virtual enterprise depicts the idea that many applications are the result of cross-organisational cooperation between several actors, playing different roles, who build a relational system which is structured by a common objective [1], [2]. Virtual enterprises can be short- or long-lived. We are mainly interested in ephemeral enterprises with the idea that they enhance the problems related to cooperation, and that what applies for short duration enterprises applies also for long duration ones. Building trade is a good example of such short-lived enterprises: it implicates a lot of partners (architect, research consultant, control office, building firm, electrician, carpenter, ...) who build an enterprise for the duration of the building work.

The objective of our work is to develop a framework to install and to support cooperation betwen the partners of a virtual enterprise, with a particular focuss on concurrent engineering applications, and especially on co-design activities. This work has been governed by two main requirements. First, partners of a virtual enterprise have not necessary a large computer men staff and, as a consequence, cooperation processes must be easy to implement. Second, partners of a virtual enterprise want to preserve their autonomy and, as a consequence, the fact that organization crosses over several partners must be as transparent as possible.

To fullfil the first requirement, we have chosen a transactional approach which releases programmers from the burden of interaction programming. To fullfil the second requirement, this approach has been implemented in a distributed manner using a cooperation model requesting a partner to negotiate cooperation patterns with its direct neighbours.

Before to deepen our transaction model (section III), we present the main concepts of our approach (section II). Section IV gives an idea of our execution framework. Finally, section V gives some trends for future work and concludes.

II. Overview of the approach

Indirect cooperation. Cooperation has a lot of dimensions and we limit ourselves to a particular point of view: cooperation by object sharing between partners. That is what we call *indirect cooperation*. Objects can be texts, plans, reports ...in different versions, from very preliminary results to attested results.

Cooperation model. In most virtual enterprises, due to the autonomy requirement, each partner has its own *local object repository* (file directories, database, ...). This local repository is composed of a *private object space* and a *cooperation object space*. Private objects can only be transferred by a local agent either in his proper *workspace* to be operated or in the *cooperation object space* to be shared with other partners. A distant partner can then transfer such cooperation objects in its own *cooperation object space* (it imports a copy). Periodically, a working agent can make visible some of his (intermediate) results by transferring an object from his *workspace* to his partner *cooperation object space*.

Cooperation patterns. One important hypothesis in this work is that these transfers follow rules which

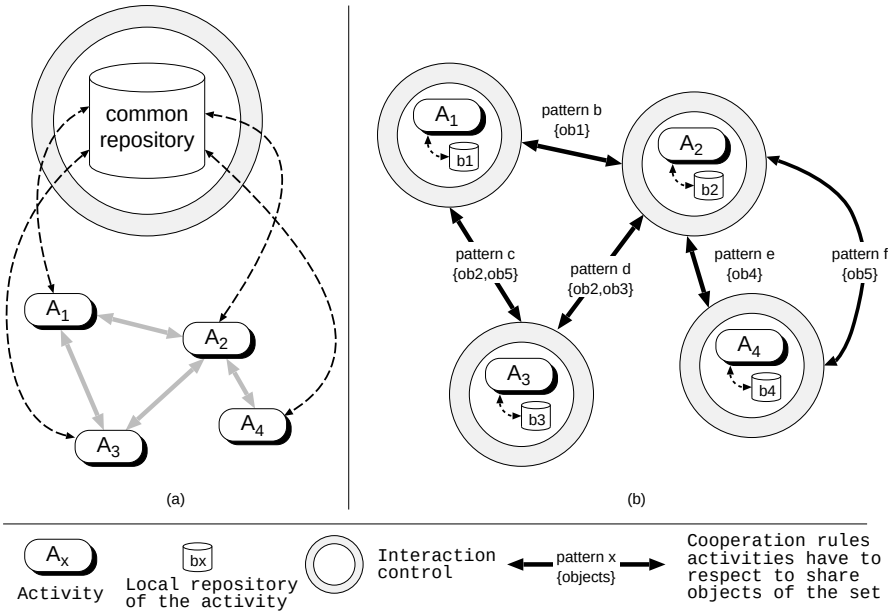

Fig. 1. Centralized vs. Peer to Peer Architecture

repeat inside one application and from one application to another. More precisely, depending on the partners roles and of (the) transferred object(s), different rules can apply. Such a set of rules is called *a cooperation pattern*. To illustrate this idea more concretely, let us take some situations that repeats during concurrent engineering activities and that we feel representative of cooperation by exchanging objects.

The *server-consumer* pattern corresponds to the case in which a partner, hereinafter called the *consumer* reads (does not write) intermediate results of another partner, the *server*. The two partners work in parallel and these exchanges can occur one or several times before the server terminates. For example, this situation occurs when a partner edits an object which includes a section written by another partner and refresh periodically its view of this section.

The *cooperative write* pattern corresponds to the case in which two partners modify two copies of the same object at the same time and exchange values of this object when working. The two partners involved in a cooperative write relationship are simultaneously server and consumer for one another for the same object.

The *server-reviewer* pattern corresponds to the case in which a partner produces an object which is reviewed by a second partner. This review is in turn read by the first partner which takes into account this review and can produce a new version of his object in response to his review. In this case, server/consumer relationships form a cycle, like in the cooperative write paradigm, but involves two different objects.

To conclude this short list, let us point out that we consider *concurrency* as a degenerated case of cooperation in which partners cannot share intermediate results, i.e. work in isolation.

It is clear that these patterns do not represent all the cooperation situations but our experience shows that they cover a large set of situations when they are combined. Moreover, current work aims to define a taxonomy of patterns based on taxonomy criteria, in relation with users in the domain of AEC and car manufacturing.

Cooperation architecture. A common way to organize cooperation is to centralize data storage and cooperation control, as depicted in figure 1-a. Due to the requirement of autonomy and the geographical distribution of sites, such an organization is not well suited to virtual enterprise needs. It breaks the autonomy of partners: first they become heavily dependant on some server; second this solution implies the existence of an administrator to which all partners whill have to entrust their data. The approach described in this paper aims to distribute both the storage and control to the partner sites in a peer to peer organization (cf. figure 1-b).

As introduced above, a partner is responsible to share or not an object, to designate the partner(s) with whom he wants to share this object, and the way he wants to share this object (the cooperation pattern

he wants to follow). More precisely, when two partners want to share an object, they first have to negotiate a cooperation pattern. Then, for each transfer operation invoked between these partners and related to this object, both of them check that all the rules fixed by the cooperation pattern are satisfied with regard to their own repository state. If both partners agree on this exchange, the transfer operation between their respective local repositories is done, otherwise this operation is denied. In that way, each partner controls himself his interactions with the other partner. Following this approach, a virtual enterprise is a graph of partners where edges are the cooperation patterns negotiated between partners to coordinate their exchanges(cf. figure 1-b).

An advantage of such a peer-to-peer architecture is its scalability: as there is no more centralized site, bottlenecks are avoided.

To conclude this overview, our approach can be summarizes in three points: *cooperation* (partners can exchange intermediate results), *distribution* (decentralization of interaction control, autonomy of partners), *negotiation* (partners can negotiate cooperation patterns).

III. A TRANSACTIONAL APPROACH

Within such a cooperation context in virtual enterprises, it seems difficult for a programmer or a set of programmers to have a global view of the whole application and to explicitly program all the interactions between activities: it is necessary to release programmers from the burden of interaction programming. In other terms, it must be possible to program a large part of cooperating activities independently of each other: application programmers should be concerned with the behavior of each activity individually, not with the interactions with other activities. In relation with these remarks, a concurrency control approach is better suited to this class of applications than a concurrent programming one. Thus, correctness of cooperative executions is based on a correctness criterion in the spirit of criteria defined for concurrency control purposes, i.e a criterion which is as much as possible not depending on the semantics of the application being synchronized [3], [4]. Another argument in favor of the concurrency control approach is that, on one hand, due to their uncertainty, it is not possible to assert correctness of executions of cooperative applications *a priori*; on the other hand, due to their long duration, it is not possible to do it *a posteriori*: it must be done incrementally.

Following a transactionnal approach as described in [5], a cooperative application is viewed as a set of transactions accessing *at the same time* a set of objects. Each activity of the application is encapsulated in a transaction that hides problems of concurrent accesses to shared objects. A transaction is then represented by the sequence of operations invoked on shared objects. However, the way of synchronizing these transactions is more complex than in classical transactional systems (administration, banking, ...) [6] in which all activities are mainly concurrent: they execute in isolation and are unaware of the others. In the context of virtual enterprises, one should rather speak about a *cooperation control* approach than a *concurrency control* approach. In the same way than [7], [8], a new extended transaction model was defined that could support such distributed cooperative applications [9], [10].

Local view. As explained in section II, the first step toward a greater autonomy of transactions is to provide each of them with a local repository in which it keeps copies of shared objects. With regard to classical transaction models where all transactions are assumed to access a common repository, a new concept was specified within our transaction model: the concept of *local repository*. Whereas classical models expressed $p_{t_i}[ob]$ in ACTA formalism [11] to denote the object event corresponding to the invocation of the operation p on object ob by transaction t_i, the same operation will be represented by $p_{t_i}[ob_{t_j}]$ in our model, where ob_{t_j} identifies the copy of the logical object ob on which the operation is invoked. ob_{t_j} denotes the copy stored in the local repository of the transaction t_j.

In that way, interactions between transactions occur through explicit data transfers beetween their respective local repositories. Such a *transfer operation* allows two transactions to synchronize their copies of the same logical object. A transfer operation is defined as a sequence of two operations, typically a *read* operation to get the value of the source copy and a *write* operation to set the value of the target copy. For instance, $(read_{t_2}[ob_{t_1}], write_{t_2}[ob_{t_2}])$ means that the transaction t_2 imports the object ob from the transaction t_1. Another way to synchroize ob_{t_2} with ob_{t_1} is to let the transaction t_1 export the object ob toward the transaction t_2, i.e. $(read_{t_1}[ob_{t_1}], write_{t_1}[ob_{t_2}])$.

Using transfer operations, a third notion could be defined: the *local history* for a transaction. It corresponds to the sequence of events (operations invoked on local/remote objects and transaction management primitives) which are relevant for this transaction. Object events logged by the local history of the transaction t_i, denoted $View_{t_i}$, are all the operations invoked by the transaction t_i itself (i.e. $\{p_{t_i}[ob_{t_j}] \in H_{ct}\}$) and all the operations invoked

by any other transaction on objects of the transaction t_i (i.e. $\{p_{t_k}[ob_{t_i}] \in H_{ct}\}$). This defines the local view of the whole system for the transaction t_i. Therefore, an operation $p_{t_i}[ob_{t_j}]$ will be logged in the local history of both transactions t_i (operation invoked by t_i) and t_j (object owned by t_j). Such operations will be used to synchronize the local histories of the various transactions.

Distributed control. The second step to improve the autonomy of the activities is to decentralize the control of the interactions. When two transactions want to exchange an object between their local repositories (one of them invokes a transfer operation), the goal is to avoid that transactions have to ask the permission to some server responsible for the control of all the interactions between all the transactions. As explained in section II, each transaction should be responsible for (i.e. to control) its own exchanges with other transactions. In other words, new properties on these exchanges have to be defined so that each transaction can ensure locally, that means using informations logged in its local history only, the same behavior of the whole system than centralized controls.

Starting from an existing correctness criterion, the *COO*-serializability[1] [12], formalized by way of axioms on the whole global history, a new distributed correctness criterion was provided: the *DisCOO*-serializability. It is defined as a set of cooperation rules (axioms) to be verified by each transaction on its local history. When these rules are satisfied by all the transactions (the distributed execution is said *DisCOO*-serializable), they ensure the whole execution is correct according to the *COO*-serializability. Inversely, if the global execution is not *COO*-serializable, then it exists a transaction whose local history breaks one of the axioms of the *DisCOO*-serializability.

The main advantage of this approach is that it is no longer needed to build the global history (using a central site or full replication mechanisms) to control the interactions between transactions. Within a classical transaction model, transactions have to connect to the common repository to invoke operations on objects. This centralization makes it easier, at the expense of transaction autonomy, to synchronize and monitor the overall execution as all controls are performed on this server which can build the global history of the whole system[2]. Using this new advanced transaction model, when an operation $p_{t_i}[ob_{t_j}]$ is invoked (eg: within a transfer operation), each of transactions t_i and t_j verifies axioms of the *DisCOO*-serializability with regard to its own local history. If they both agree, the new event is accepted and logged into their local histories. Otherwise, the event is rejected. Transactions t_i and t_j are the only ones to be aware of this event and they can take themselves the decision of acceptance for this event. So they can cooperate by exchanging data even if they are cut off from the outside world; they only need to be connected together.

Such a peer-to-peer architecture is well suited for transaction autonomy. However, the impact on the axiomatic definition of the correctness criteria is significant. Each transaction should be able to evaluate these axioms using informations logged in its local history only. Particularly, that means it is not possible to detect, on this local view of the system, any cycle in the graph of transaction dependencies. And the definitions of classical correctness criteria, like the *COO*-serializability or the serializability, are precisely based on the notion of cycle of dependencies between transactions. So it is mandatory to define new properties on local histories that ensure, step by step, the same behavior of transactions than properties based on cycles.

For instance, whereas the *COO*-serializability sets a dependency between transactions t_c and t_s when t_c reads an intermediate result from t_s, defines groups of transactions, and imposes that all transactions within a group reach a common state before they commit (such a property must be defined on the whole global history), the *DisCOO*-serializability only requires for a transaction to be up to date with regard to all transactions from which it read some result before it can commit. That means that if a transaction t_c imported an object *ob* from a transaction t_s, t_c must have imported the last version of *ob* produced by t_s when t_c want to commit. Such a property is easily evaluated on the local history of each transaction t_s used in the local history of t_c. If ever a situation occurs where the *COO*-serializability should have grouped a set of transactions, the property *up_to_date* of the *DisCOO*-serializability will ensure, step by step, that all these transactions are up to date one with regard to the others before they can commit, and consequently that they agree on the final state of shared objects.

[1] The *COO*-serializability allows transactions to share intermediate results during their execution, but ensures that if a transaction reads an intermediate result, then this transaction will read the corresponding final result before it commits. If a cycle between transaction dependencies is detected, all the transactions implied in the cycle are grouped, they have to reach a common state, and they will commit at the same time.

[2] Specifically, when a transaction invokes an operation, the preconditions of this event derived from the axiomatic definition of the transaction model are evaluated with respect to the current history. If its preconditions are satisfied, the new event is accepted and appended to the current history. Otherwise, the event is rejected.

Cooperation schemas. Following this approach, transactions are really autonomous, both for the accesses to shared objects (each transaction owns a copy) and for the interaction control (each transaction is responsible for its own exchanges with other transactions). Within such a context, the next step is to allow *DisCOO*-transactions to negotiate the cooperation rules to be checked for their exchanges. In other words, when two transactions t_i and t_j want to share an object *ob*, they first have to negotiate a cooperation schema *sch*. Such a cooperation schema is a set of cooperation rules (or local properties) and corresponds to a given correctness criterion of data exchanges between two transactions. This negotiation is denoted by the event $Contract[t_i, t_j, ob, sch]$ appended to the current local history of each transaction t_i and t_j. Then, whenever a transfer operation for this object is invoked between these two transactions (containing either $p_{t_i}[ob_{t_j}]$ or $p_{t_j}[ob_{t_i}]$), both t_i and t_j will verify that all properties of the cooperation schema *sch* are satisfied with regard to its local history.

In this way, this new advanced transaction model is not linked with a given (complex) correctness criterion that should be defined to support all kinds of interactions between all the transactions of the system. On the contrary, it provides mechanisms to configure the *"interaction control"* component of the figure 1-b with various cooperation schemas (viewed as *basic correctness criteria*) negotiated between *DisCOO*-transactions. Within this transaction model, the concept of negotiation is formalized by adding new axioms on events $Contract[\ldots]$. For instance, one axiom imposes that both transactions t_i and t_j log the event $Contract[t_i, t_j, \ldots]$ in their local hitory at the same time. As explained above, another axiom ensures that transactions t_i and t_j negotiated a cooperation schema *sch* for the object *ob* and that *sch* is satisfied before the invocation of an operation like $p_{t_i}[ob_{t_j}]$.

This new advanced transaction model was designed to support the cooperation between distributed activities as described in figure 1-b. We focused this paper on our main requirement: the autonomy of these activities. By *autonomy* we mean that each activity owns a local repository, is responsible for the control of its exchanges with other activities, and can negotiate the cooperation rules with its partners.

IV. Implementation

This new advanced transaction model was put into practice within the prototype *DisCOO*. This middleware application aims to manage and coordinate object sharing between distributed activities. *DisCOO* provides each activity with three main components (cf. figure 2): a *cooperation object space*, a *private workspace*, and a *coordinator*. The cooperation object space is the public part of the local repository of the activity. The workspace allows us to view objects of the cooperation space as files and directories so that an agent can use its legacy applications (Word, Autocad, emacs,...) to work on shared objects (cf. the cooperation model description in section II). When an agent want to publish its work, it updates modified objects in its cooperation space. Thus, updates are made visible to partners which can import them within their own cooperation space. Finally, the most interesting component is the coordinator that denotes the *"interaction control"* component shown in figure 1-b. All object exchanges between cooperation spaces of activities have to occur through their respective coordinator. For an activity, its coordinator is responsible for storing cooperation schemas negotiated by this activity with its partners and for ensuring that all operations invoked wih regard to this activity satisfy these cooperation schemas (cf. the distributed control in section III).

Within the prototype *DisCOO*, activities are connected together on the Internet by the use of a CORBA[3] software object bus. Thus *DisCOO* is really based on a peer-to-peer architecture as described in figure 1-b. For ease of deployment, all *DisCOO* components were developed in Java, so the same code is used to connect activities running on Windows 98/NT computers with activities running on Solaris/Linux/... stations. The last part, but not the least, concerns the cooperation schemas and the way they are verified. Each of them was formally defined as a set of ACTA axioms (first ordre logic formulae). Instead of developping specific protocols, we rather chose to translate these ACTA axioms in Prolog predicates evaluated on local histories. Thus new cooperation schemas can easily be defined for use within *DisCOO*.

V. Concluding remarks

The transactional approach we presented in this paper releases programmers from the burden of interaction programming for cross-organizational cooperation support. Compared to classical transaction models, our *DisCOO*-transaction model introduces three innovations: *cooperation* (activities can exchange intermediate results during their execution), *distribution* (decentralization of interaction control, cooperation rules evaluated on local histories, autonomy of activities), *negotiation* (activities can negotiate coop-

[3]Common Object Request Broker Architecture

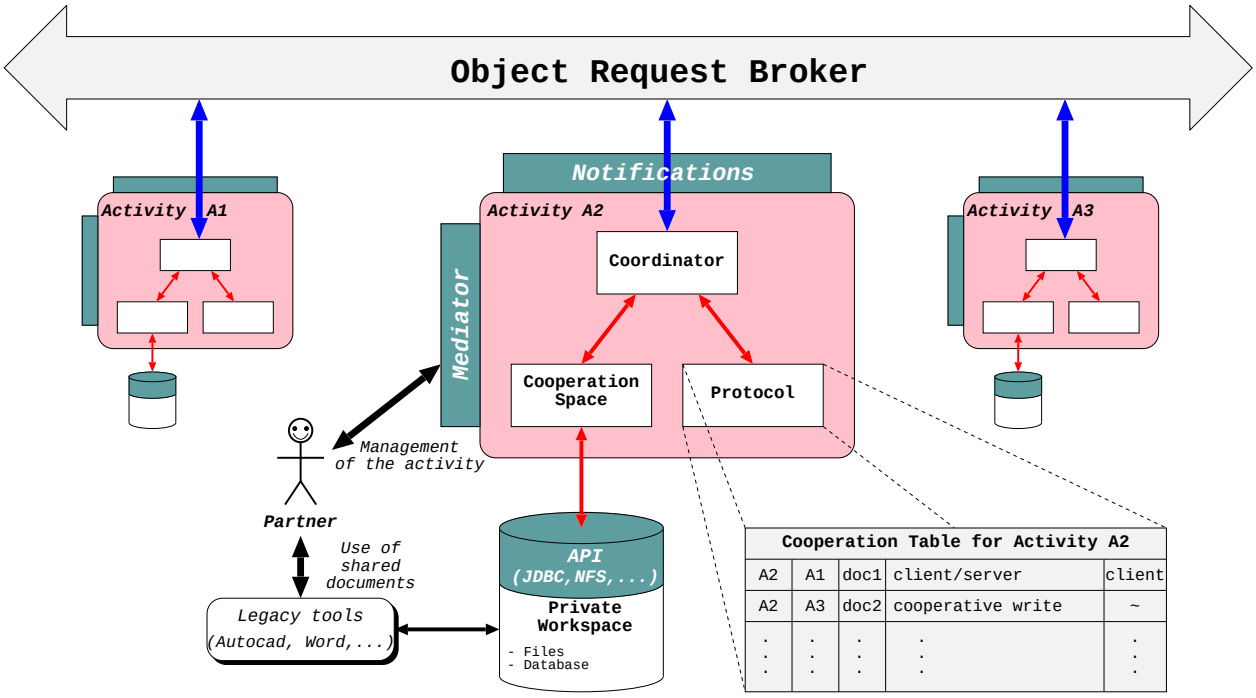

Fig. 2. Architecture of the prototype *DisCOO*

eration schemas to share objects). Further details on this work can be found in [13].

One of our prospects is to integrate the negotiation step within our model. For now, our framework provides activities with cooperation support from the time the *Contract*[...] event, i.e. the result of the negotiation, is logged in their local histories. We are working for formalizing this negotiation step at the same level than the other operations. Doing this, we aim to better support later cooperation schema negotiation or conflict resolution.

Another prospect is to analyse cooperation behaviour to deduce new cooperation schemas that we can formalize as correctness criteria and implement within our prototype *DisCOO*. The paper [14] presents some early work in this way.

References

[1] M. Hardwick and R. Bolton, "The industrial Virtual Enterprise," *Communications of the ACM*, vol. 40, no. 9, September 1997.

[2] M. Ribière and N. Matta, "Virtual Enterprise and Corporate Memory," in *Building, maintaining and using organizational memories workshop of ECAI'98*, Brighton, August 1998.

[3] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing surveys*, vol. 13, no. 2, pp. 186–221, 6 1981.

[4] K. Ramamritham and P.K. Chrysanthis, "A taxonomy of correctness criteria in database applications," *The VLDB Journal*, vol. 5, no. 5, pp. 85–97, 1996.

[5] P.A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann, 1997.

[6] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[7] A.K. Elmagarmid, Ed., *Database transaction models for advanced applications*, Morgan Kauffman, 1992.

[8] S. Jajodia and L. Kershberg, Eds., *Advanced Transaction Models and Applications*, Morgan Kauffman, 1997.

[9] M. Munier and C. Godart, "Cooperation services for widely distributed applications," in *Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco Bay, USA, 1998.

[10] K. Benali, M. Munier, and C. Godart, "Cooperation models in co-design," *International Journal on Agile Manufacturing (IJAM)*, vol. 2, no. 2, 1999.

[11] P.K. Chrysanthis and K.Ramamritham, "Synthesis of Extended Transaction Models," *ACM Transactions on Database Systems*, vol. 19, no. 3, pp. 451–491, September 1994.

[12] G. Canals, C. Godart, P. Molli, and M. Munier, "A Criterion to Enforce Correctness of Indirectly Cooperating Applications," *Information Sciences*, vol. 110/3-4, pp. 279–302, September 1998.

[13] Manuel Munier, *Une architecture pour intégrer des composants de contrôle de la coopération dans un atelier distribué*, Thèse en informatique, Université Henry Poincaré - Nancy I, Loria, janvier 1999.

[14] S. Tata, G. Canals, and C. Godart, "Specifying interactions in Cooperative applications," in *Eleventh International Conference on Software Engineering and Knowledge Engineering (SEKE'99)*, Kaiserslautern , Germany, 1999.